

ALGORITHMS AND DYNAMIC DATA STRUCTURES

CHAPTER 1: INTRODUCTION TO POINTERS (04h)

Prof. Abdelfetah HENTOUT
abdefetah.hentout@enscs.edu.dz

Chapter 1: Introduction to pointers

- Introduction.
- Basic concepts:
 - Definition.
 - Declaration.
 - Initialization.
 - Operators.
 - Usage.
- Arithmetic on pointers.
- Pointers and arrays.
- Pointers, Characters and Strings.
- Static and dynamic allocation.

Introduction (1)

The memory space allocated to a running program is made up of two main parts:

- **Code part:**
 - Reserved to contain the program code.
- **Data part:**
 - Reserved to store the data manipulated by the program.

```
Procedure Example;  
Variables x, i: Integer;      c: Char;  
           Tab: Array[6] of Char;  
           ...  
Begin  
    ...  
    x  $\leftarrow$  15;  
    c  $\leftarrow$  'a';  
    For(i  $\leftarrow$  1; i  $\leq$  6; i  $\leftarrow$  i+1)    Tab[i]=getASCII(65+i);  
    ...  
End;
```

Introduction (2)

x	15
i	1
c	'a'
Tab[1]	ASCII(65+1)
Tab[2]	ASCII(65+2)
Tab[3]	ASCII(65+3)
Tab[4]	ASCII(65+4)
Tab[5]	ASCII(65+5)
Tab[6]	ASCII(65+6)
...	...
...	...
@1	$x \leftarrow 15;$
@2	$c \leftarrow 'a';$
@3	For ($i \leftarrow 1; i \leq 6; i \leftarrow i+1$)
@4	Tab[i]=getASCII(65+i);

Data
part

Code
part

Memory area associated with the algorithm:

- Variables are used to store data.
- The variable value is located in a specific location (slot) in the internal memory.
- The variable identifier allows to directly access its value.

Introduction (3)

During its lifetime, each declared variable is associated with:

- An identifier.
- A memory area reserved for it:
 - Identified by the **start address** and its **size**.
- Each declared variable is associated with an area (slot) in the main memory.

```
Variables c: Char;  
           x: Integer;  
           Tab: Array[10] of Char;
```

Begin

```
...  
c ← 'z';  
Tab ← {'a', 'b', 'c', 'd', 'e'};  
x ← 8970;
```

...

End;

Introduction (4)

c

'z'

0x100

- Start address of variable **c** = 0x100.
- Size = 1 byte.

x

8970

0

0

0x104

0x105

0x106

0x107

- Start address of variable **x**: 0x104.
- Size = 4 bytes.

Tab

97

98

99

100

101

...

-

-

0x108

0x109

0x110

0x111

0x112

...

0x116

0x117

- Start address of variable **Tab** = 0x108.
- Size = 10 bytes.


Basic concepts (1)

- A pointer is a **special variable** that **stores memory addresses**:
 - It is limited to a single data type.
- It may contain the address of:
 - A simple variable.
 - A complex type.
 - An array.
 - etc.
- Pointers are mainly used to **indirectly** manipulate:
 - Other variables, or
 - Memory areas.
- Pointers are a powerful low-level programming tool.
 - But, a dangerous low-level programming tool.

Basic concepts (2)

Pointers vs. Variables

- Pointers and variable identifiers have the same role:
 - Provide access to an internal memory slot.
- It is important to distinguish:
 - A pointer is a variable that can **point** to different addresses:
 - $\text{ptr} \leftarrow @1;$
 - $\text{ptr} \leftarrow @2;$
 - $\text{ptr} \leftarrow @3;$
 - A variable always remains associated to the same address.



x	15
y	300
find	TRUE
ptr	@3
...	...
...	...
...	...
...
@1	??
@2	??
@3	??

Basic concepts (3)

Two main addressing modes:


- **Direct addressing**

- In programming, variables are used to store data.
 - The value of a variable is located at a specific slot in the memory.
- The variable identifier allows directly accessing the value.

- **Indirect addressing**

- If not possible to use a variable identifier:
 - Copy the address of this variable into a special variable called a **pointer**.
 - Retrieve the data of the variable by going through the **pointer**.

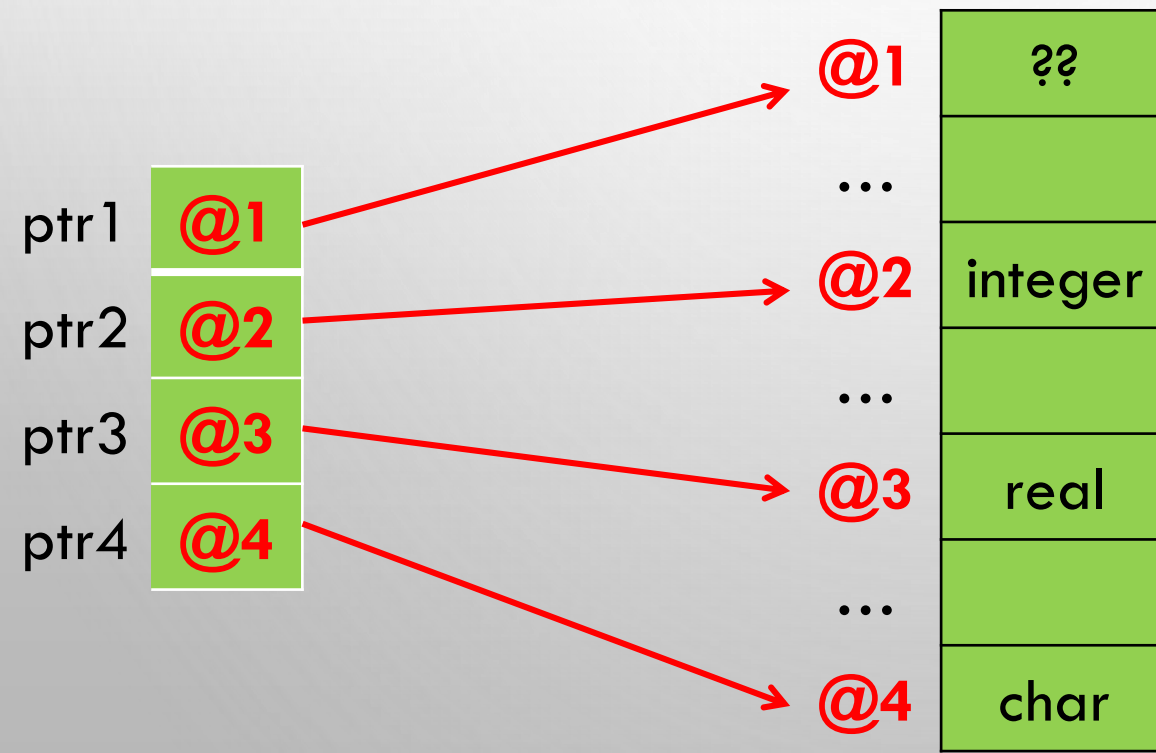
x	15
y	300
find	TRUE
Tab[0]	'a'
Tab[1]	'b'
Tab[3]	'c'
...	...
...	...
ptr1	@1
ptr2	@2



Basic concepts (4)

Pointers are variables of type address:

```
ptr1: *Void;           // (void pointer) ptr1 can hold any address.  
ptr2: *Integer;        // ptr2 holds only addresses of integers.  
ptr3: *Real;           // ptr3 holds only addresses of reals.  
ptr4: *Char;           // ptr4 is supposed to hold only addresses of chars.
```



Basic concepts (5)

To initialize a pointer, i.e. to give it a default value:

- Do not use the number **0**.
- Use instead the keyword **NULL** (in capital letters).

Algorithm PointerExample;	
Variables myPointer: *Integer; pointerToAge: *Integer;	These lines reserve two slots in the memory to store two addresses (myPointer , pointerToAge)
age: Integer;	This line reserves a slot in the memory to store the variable age
Begin	
...	
myPointer ← NULL ;	The slot does not contain any address at the moment (NULL).
age ← 10;	This line assigns the value 10 to age .
pointerToAge ← &age;	This line assigns the address of the variable age to pointerToAge .
...	
End;	


Basic concepts (6)

Do pointers have a type?

- There is **no type** for **pointer**, such as Integer, Real, Boolean, etc.
 - Do not write **Pointer** pointerToAge.
 - Instead, use the symbol *:
 - Indicate the type of the variable on which will point the variable.

```
Algorithm Example;  
Variables age: Integer;  
        pointerToAge: *Integer;  
Begin  
    ...  
    age ← 10;  
    pointerToAge ← @age ;  
    ...  
End;
```

Variable	Address	Value
	0x0	145
	0x1	3
	0x2	82
pointerToAge	0x3	0x745
...
...
age	0x745	10



Basic concepts (7)

- Since the variable **pointerToAge** points on the variable **age** (which is **Integer**):
 - The pointer must be ***Integer**.
- If the variable **age** is Real:
 - The pointer must be ***Real**.

Algorithm Example;

Variables age: Integer;

 pointerToAge: *Integer;

Begin

 age \leftarrow 10;

 pointerToAge \leftarrow @age ;

 print(pointerToAge);

0x745

 print(*pointerToAge);

10

 print(@pointerToAge);

0x3

End;

Variable	Address	Value
	0x0	145
	0x1	3
	0x2	82
pointerToAge	0x3	0x745
...
...
age	0x745	10

Basic concepts (8)

- The variable identifier always remains linked to the same address.
 - **age**: I need the value of the variable **age**.
 - **@age**: I need the address at which the variable **age** is located.
- A pointer is a variable that can point to different addresses.
 - **pointerToAge**: I need the value of **pointerToAge** (this value is being an address).
 - ***pointerToAge**: I need the value of the variable that is located at the address pointed by **pointerToAge**.

```
Algorithm Example;  
Variables age: Integer;  
          pointerToAge: *Integer;  
Begin  
    age ← 10;  
    *pointerToAge ← @age ;  
    print(pointerToAge);  
    print(*pointerToAge);  
    print(@pointerToAge);  
End;
```

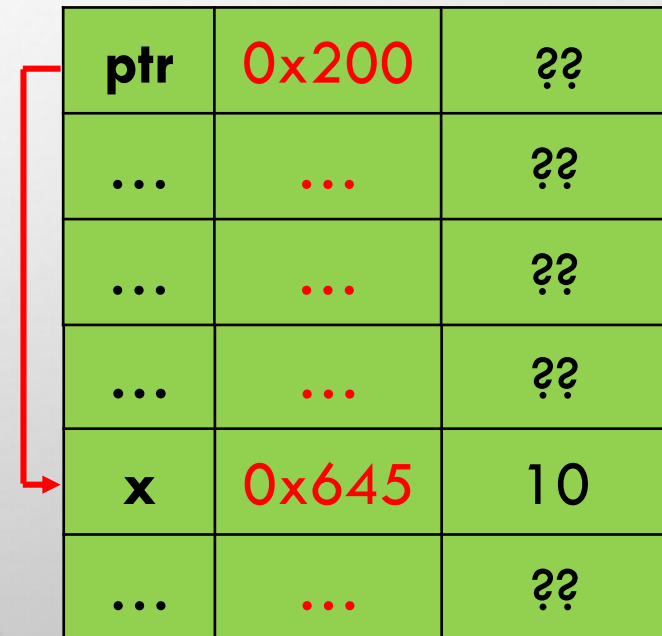
Basic concepts (9)

When using pointers, we need:

- Operator “**Address of**” to get the address of a variable:
 - Algorithm: @.
 - C/C++: &.
- Operator “**Content of**” to access the content of an address (pointer):
 - Algorithm/C/C++: *.

Example:

- **ptr**: Uninitialized pointer.
- **x**: Variable (of the same type):
 - $x \leftarrow 10$.
- $\text{ptr} \leftarrow @x$ ($\text{ptr} = \&x$) assigns the address of variable **x** to variable **ptr**.
 - **ptr** points **x**



ptr	0x200	??
...	...	??
...	...	??
...	...	??
x	0x645	10
...	...	??

Basic concepts (10)

Variables `x: Integer;` `// x is a variable of type integer`
`ptr: *Integer;` `// ptr is a variable of type pointer to integer`



`ptr ← &x;` `// assign to ptr, the address of x, (i.e., ptr points x)`
`// actually, ptr points the slot located at address 0x100`



`*ptr ← 10;` `// assign the value 10 to the slot pointed to by ptr (i.e., to x)`



Basic concepts (11)

Designation	Algorithm	C/C++
Declaration of the pointer variables p1 and p2 , which contain the addresses of variables Type	$p1, p2: *Type;$	<code>Type *p1, *p2;</code>
Initialization to an empty address (NULL), which indicates that the pointer does not point anywhere .	$p1 \leftarrow NULL;$ $p2 \leftarrow NULL;$ $(p1 \leftarrow p2 \leftarrow NULL;)$	<code>p1=NULL;</code> <code>p2=NULL;</code> <code>(p1=p2=NULL;)</code>
Assignment operator	$p1 \leftarrow p2;$	<code>p1=p2;</code>
Operator Address of , returns the address of a variable x	$p1 \leftarrow @x;$	<code>p1=&x;</code>
Operator Content of , accesses the memory slot pointed by p	$*p \leftarrow x;$ $(a \leftarrow x;)$	<code>*p = x;</code> <code>(a = x;)</code>

Passing parameters modes (1)

Recall

- Passing parameters is a very important concept.
- Two different modes of passing parameters:
 - **Keeping the value of an input parameter after module execution:**
 - A **passing parameter by value** must be performed.
 - **Taking the new value of an input parameter after module execution (modified value of the parameter):**
 - A **passing parameter by variable (by address or by reference)** must be performed.
 - The word **VAR** must **precede** the **formal parameter**.

Passing parameters modes (2)

Recall: Solution of a second degree equation

Procedure SecDegSol(a, b, c: Real; **Var** nSol: Integer; **Var** x1, x2: Real);

Variables delta: Real;

Begin

delta \leftarrow b*b-4*a*c; // delta \leftarrow SQR(b)-4*a*c;

IF(delta<0) nSol \leftarrow 0; // No solution

Else // delta \geq 0

Begin

If(delta>0) // Two solutions

Begin

nSol \leftarrow 2;

x1 \leftarrow (-b- $\sqrt{\text{delta}}$)/(2*a); // x1 \leftarrow (-b-SQRT(delta))/(2*a);

x2 \leftarrow (-b+ $\sqrt{\text{delta}}$)/(2*a);

End;

Else // delta=0

Begin // One solution

nSol \leftarrow 1;

x1 \leftarrow x2 \leftarrow -b/(2*a);

End;

End;

End;

Passing parameters modes (3)

Passing parameter by variable (by address or by reference)

- In C/C++ language, pointers realize the **by variable passing mode** of the input/output parameters of the module (function or procedure)

Designation	Algorithm	C/C++
Prototype	Procedure SecDegSol(a, b, c: Real; Var nSol: Integer; Var x1, x2: Real);	void SecDegSol(float a, float b, float c, int *nSol, float *x1, float *x2);
Assignment	$nSol \leftarrow 2;$ $x1 \leftarrow (-b - \sqrt{\text{delta}}) / (2 * a);$ $x2 \leftarrow (-b + \sqrt{\text{delta}}) / (2 * a);$	$*nSol = 2;$ $*x1 = (-b - \text{sqrt}(\text{delta})) / (2 * a);$ $*x2 = (-b + \text{sqrt}(\text{delta})) / (2 * a);$
Call/Invoke the procedure	SecDegSol(a, b, c, @nSol, @x1, @x2);	SecDegSol(a, b, c, &nSol, &x1, &x2);

Passing parameters modes (4)

Passing parameter by variable (by address or by reference)

- In C/C++ language, pointers realize the **by variable passing mode** of the input/output parameters of the module (function or procedure)

```
void triplePointer(int *pointerToNumber);  
int main (){  
    int num = 5;  
    triplePointer(&num);    // send the address of num to the function  
    printf ("%d", num);    // display the variable num  
    return 0;  
}
```

```
void triplePointer(int *pointerToNumber){  
    *pointerToNumber *= 3;    // multiply the value by 3  
}
```

Passing parameters modes (5)

```
void triplePointer(int *pointerToNumber);  
int main (){
```

```
    int num = 5;
```

```
    triplePointer(&num);
```

```
    printf ("%d", num);
```

```
    return 0;
```

```
}
```

Create **num** variable in the main function.
Initialize **num** = 5.

Call the function **triplePointer**
Send the address of **num** as an input parameter.

Back into the main function, **num** = 15.
Function **triplePointer** has directly modified **num**

```
void triplePointer(int *pointerToNumber){  
    *pointerToNumber *= 3;  
}
```

triplePointer receives the address of **num** in **pointerToNumber**.

Having the **pointer** to **num**, directly modify **num** in its memory slot.

Simply use ***pointerToNumber** to designate **num**.

Passing parameters modes (6)

Example: Using a pointer to Point2D, write the following modules:

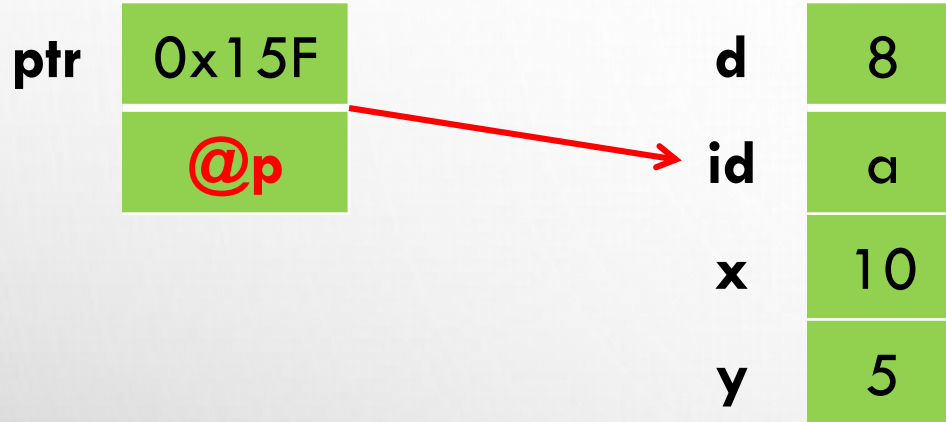
- Display the coordinates of a point.
- Read the coordinates of a point.
- Initialize the pointer to NULL.
- Modify the coordinates of a point.

Algorithm	C/C++
TYPE Point2D = Record Begin id: Char; x, y: Real; End;	typedef struct Point2D{ char id; float x, y; }Point2D;

Passing parameters modes (7)

Desig.	Algorithm	C/C++
Display the coordinates of a point	Procedure printP(p: *Point2D) Begin print(p.id, "(", p.x, " , ", p.y, ")"); End;	void printP(Point2D *p){ print("%c(%d, %d)", p.id, p.x, p.y); }
Read the coordinates of a point	Function readP(): *Point2D variables p: Point2D; begin read(p.id); read(p.x); read(p.y); return(@p); end;	Point2D* readP(){ Point2D p; scanf("%c", &p.id); scanf("%d", &p.x); scanf("%d", &p.y); return(&p); }
Initialize the pointer to NULL	Procedure Init(Var p: *Point2D); Begin p ← NULL; End;	void Init(Point2D* *p){ *p=NULL; }
Modify the coordinates of a point	Procedure Modify(Var p:Point2D, d:Integer); Begin p.x ← p.x + d; p.y ← p.y + d; End;	void Modify(Point2D *p, int d){ *p.x=*p.x+d; //p->x=p->x+ d; *p.y=*p.y+d; //p->y=p->y+d; }

Passing parameters modes (8)



```
void Modify(Point2D *p, int d){  
    *p.x=*p.x+d;    //p->x=p->x+ d;  
    *p.y=*p.y+d;    //p->y=p->y+d;  
}
```

d	8
id	a
x	18
y	13

Dynamic memory allocation (1)

How to deal with an unknown number of values?

- In case the maximum number of elements is known (or can be estimated):
 - Use arrays.
- This solution is limited because:
 - Waste memory space.
 - Inadequate:
 - In case the maximum number of elements changes frequently:
 - ☐ Scholarship promotions of different years.
 - ☐ Recent baccalaureate students.
 - ☐ etc.

Dynamic memory allocation (2)

Solution:

- **Declare elements only when needed.**
 - Array type does not allow this, being a static data structure.
 - Array size can neither be incremented nor decremented.
- **Dynamic allocation and release mechanism:**
 - Slots are reserved if necessary.
 - Slots are released if not needed.
- **New dynamic data structures:**
 - Linked linear lists
 - Trees.



Object of next lectures

Dynamic memory allocation (3)

Dynamic allocation vs. Static allocation

Designation	Static allocation	Dynamic allocation
Size of the memory space	Static (fixed)	Dynamic (variable), incremented or decremented
Space allocation is done	At the beginning of the program	Gradually during the running of the program
Memory space is allocated	At compilation	During execution
Appropriate data structure	Arrays	Linked linear lists Trees

Dynamic memory allocation (4)

Advantages and drawbacks of Dynamic allocation

Advantages	Drawbacks
<ul style="list-style-type: none">• End of a programming era where everything is defined and fixed entirely in advance• Adaptation to the environment: Not all computers are identical (memory capacities, etc.)• Native management of some dynamic data structures (arrays, etc.)	<ul style="list-style-type: none">• Lack of simplicity compared to static data structure management• Necessity to master pointers• Monitor memory allocation and release at the end of its use (otherwise risk of "leaks")• Copy a dynamic structure is harder a static structure (code, resources)• Exception management during allocation and release

Dynamic memory allocation (5)

Dynamic structures are recognized as being the source of many risks:

- Often, objects implement data dynamically.
 - It is appropriate to configure/control such a structure to avoid memory saturation/overflow problems or outside access.
- Premature termination of the process using a dynamic structure:
 - Bad implementation.
 - No protection.
- Blocking of the operating system on which the process is running.
- Implementation of permanent control of the dynamic structure until its destruction.

Dynamic memory allocation (6)

Tips:

- Initialize a pointer to NULL when declared.
- Set the pointed data to **zero** as soon as it is no longer used.
- Reset the pointer to NULL as soon as:
 - No longer useful.
 - Before stopping the program.
 - Before destroying it.

Variables ptr: *Integer; // ptr is a variable of type pointer to integer

...

Begin

ptr = NULL;

ptr=malloc(sizeof(int));

*ptr = 0;

...

free(ptr);

End;

Dynamic memory allocation (7)

How to create and destroy variables during the execution of programs using predefined functions?

- In C/C++, the standard library contains some functions for dynamic memory allocation (**#include <stdlib.h>**):
 - **malloc(n)**: allocate a contiguous block of **n** bytes and returns its address (i.e., the start address of the allocated block).
 - **free(p)**: frees and deallocates the specified block of memory (**p**).
- It is possible to dynamically add any number of variables during the program running.
- The new variables have no identifiers
 - They can only be manipulated through their addresses.
 - Via pointers.

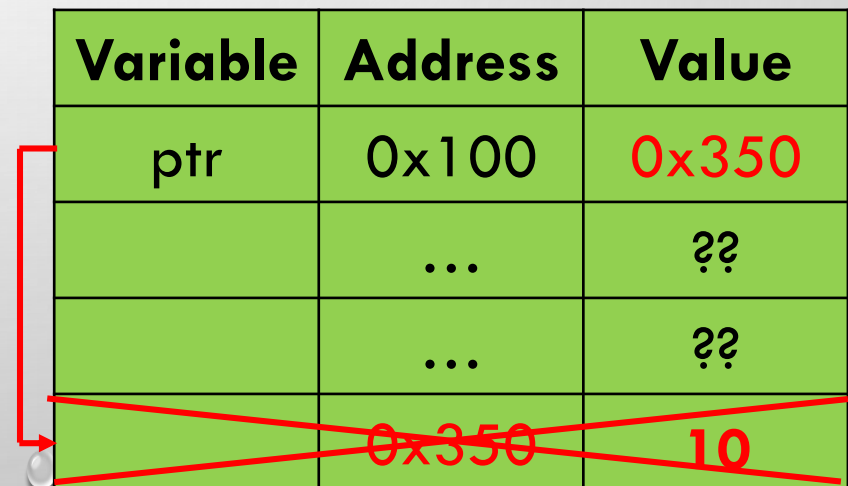
Dynamic memory allocation (8)

Example:

```
int *ptr;  
ptr = malloc(sizeof(int));    // a new dynamic variable is allocated  
// ptr contains now the address of this new variable.  
// New variable can be used like any int type variable and can  
// be manipulated indirectly via any pointer containing its address
```

```
*ptr = 10; // indirectly assign 10 to  
this new dynamic variable
```

```
free(ptr); // release (destroy) the  
dynamic variable pointed to by ptr
```



Variable	Address	Value
ptr	0x100	0x350
	...	??
	...	??
		0x350 10

Dynamic memory allocation (9)

Example 2:

```
struct enreg{
    int score;
    char Tab[5];
    double *count;
}*ptr;
ptr=malloc(sizeof(struct enreg));

(*ptr).score=10;    //ptr->score=10;
(*ptr).Tab[0]='a';  //ptr->Tab[0]='a';
(*ptr).Tab[1]='b';  //ptr->Tab[1]='b';
(*ptr).Tab[2]='c';  //ptr->Tab[2]='c';
(*ptr).Tab[3]='d';  //ptr->Tab[3]='d';
(*ptr).Tab[4]='e';  //ptr->Tab[4]='e';

(*ptr).count=malloc(8*sizeof(double));
// ptr->count=malloc(8*sizeof(double));

(*ptr).count[3]=3.14;    //ptr->count[3]=3.14;
//ptr->count+3=3.14; //*((*ptr).count+3)=3.14;
(*ptr).count[5]=2.71;
```

ptr	0x100	0x350
	...	??
score	0x350	??
Tab[0]		??
Tab[1]		??
Tab[2]		??
Tab[3]		??
Tab[4]		??
count		??
	...	
	0x600	??
		??
		??
		??
		??
		??
		??

Dynamic memory allocation (10)

How to create and destroy variables during the execution of programs using predefined functions?

Designation	Algorithm	C/C++
Allocation	Function allocate(): *TypePointer;	void* malloc(size_t size); void* calloc(size_t elt_count, size_t elt_size); void* realloc(void *p, size_t size);
Release	Procedure free(Var P: *TypePointer);	void free(void *p);

Arithmetic on pointers (1)

Priority of operators * and &

- Operators * and & have the same priority as the other unary operators:
 - (!, ++, --).
- In the same expression, operators are evaluated from right to left.
- If a pointer **ptr** points to a variable **x**:
 - ***ptr** can replace **x** anywhere.

Example:

Variables x: Integer;
 ptr: *Integer;

Begin

...
ptr = @x;
...

End;

Expression	Equivalent
$y \leftarrow *ptr + 1;$	$y \leftarrow x + 1;$
$*ptr \leftarrow *ptr + 10;$	$x \leftarrow x + 10;$
$*ptr += 2;$	$x +\leftarrow 2;$
$++*ptr;$	$++x;$
$(*ptr)++;$	$x++;$

Arithmetic on pointers (2)

Two arithmetic operations can be performed on pointers:

- Addition/subtraction of Integers:**
 - ptr** points to a value of **TypeElement** located at the address **add**.
 - sizePtr** is the bytes number occupied by a **TypeElement** variable.
 - It is possible to define an expression in which we add/subtract an integer **k** to the pointer **ptr**:
 - The value of this expression is an address of the same type as **ptr**, increased/decreased by $k * \text{sizePtr}$ bytes:
 - Expression $\text{ptr} + k = \text{add} + k * \text{sizePtr}$.
 - Expression $\text{ptr} - k = \text{add} - k * \text{sizePtr}$.

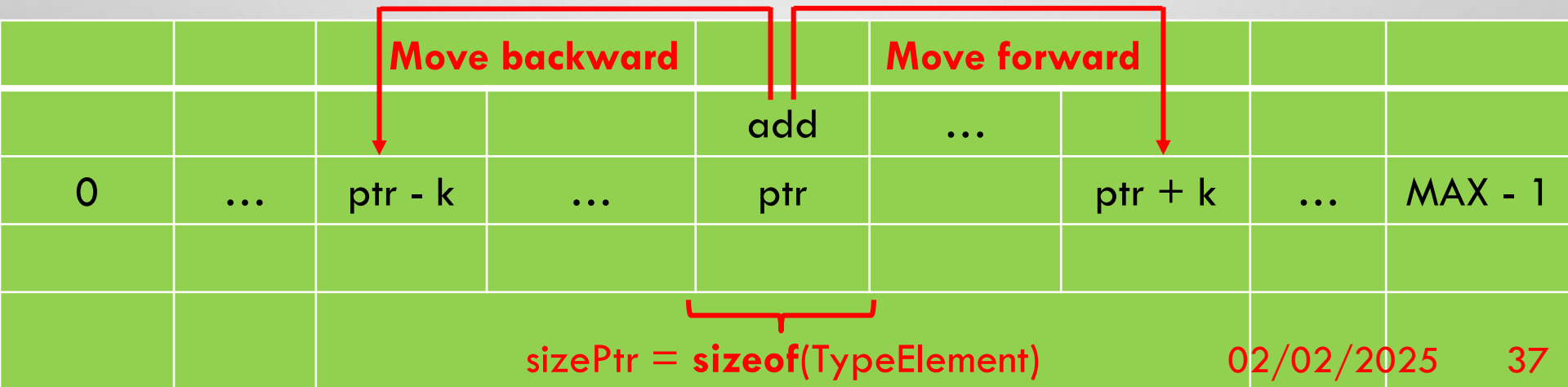
				add	...				
0	...	ptr - k	...	ptr		ptr + k	...	MAX - 1	
		sizePtr = sizeof(TypeElement)							

02/02/2025 37

Two arithmetic operations can be performed on pointers:

1. Addition/subtraction of Integers:

- **ptr** points to a value of **TypeElement** located at the address **add**.
- **sizePtr** is the bytes number occupied by a **TypeElement** variable.
- It is possible to define an expression in which we add/subtract an integer **k** to the pointer **ptr**:
 - The value of this expression is an address of the same type as **ptr**, increased/decreased by $k * \text{sizePtr}$ bytes:
 - Expression $\text{ptr} + k = \text{add} + k * \text{sizePtr}$.
 - Expression $\text{ptr} - k = \text{add} - k * \text{sizePtr}$.



Arithmetic on pointers (3)

Example 1:

Tab

Index	0	1	2	3	4	5	6	...	MAX-1
Value	v0	v1	v2	v3	v4	v5	v6	...	
@	&Tab[0]	&Tab[1]	&Tab[2]	&Tab[3]	&Tab[4]	&Tab[0]	&Tab[0]	⋮	&Tab[MAX-1]

size = MAX * sizeof(TypeElement) bytes

$\&\text{Tab}[1] = \&\text{Tab}[0] + 1 * \text{sizeof}(\text{TypeElement})$

$\&\text{Tab}[2] = \&\text{Tab}[0] + 2 * \text{sizeof}(\text{TypeElement})$

$\&\text{Tab}[3] = \&\text{Tab}[0] + 3 * \text{sizeof}(\text{TypeElement})$

...

$\&\text{Tab}[i] = \&\text{Tab}[0] + i * \text{sizeof}(\text{TypeElement})$

...

$\&\text{Tab}[\text{MAX}-1] = \&\text{Tab}[0] + (\text{MAX}-1) * \text{sizeof}(\text{TypeElement})$

Arithmetic on pointers (4)

Example 2:

```
void main(){
    int Tab[10];          int *ptr, i;
    ptr = Tab;             // same as ptr = &Tab[0]
    *ptr = 10;             // same as Tab[0] = 10
    *(ptr+1) = 20;        // same as Tab[1] = 20
    ptr[2] = 30;          // same as Tab[2] = 30
    Tab[3] = 40;
    *(Tab+4) = 50;        // same as Tab[4] = 50
    printf("Addresses are displayed in hexadecimal\n");
    printf("Iterate through array elements using formula : @Base+Offset\n");
    printf("Base_Addr(Tab) \t Offset(i) \t Contents *(Tab+%d)\n");
    for (i=0; i<5; i++)
        printf("%p \t %8d \t %12d\n", Tab, i, *(Tab+i) );
    printf("Iterate through array elements using pointer ptr incrementation\n");
    printf("Elem Addr(ptr) \t Contents (*ptr)\n");
    for (i=0; i<5; i++) {
        printf("%p \t %12d\n", ptr, *ptr);
        ptr++;             // move to next element (by step of sizeof(int) )
    }
}
```

Arithmetic on pointers (5)

Example 2: Addresses are displayed in hexadecimal

Iterate through array elements using formula : $@Base + Offset$

Base_Addr(Tab)	Offset(i)	Contents *(Tab+i)
----------------	-----------	-------------------

0x7ffe79199540	0	10
0x7ffe79199540	1	20
0x7ffe79199540	2	30
0x7ffe79199540	3	40
0x7ffe79199540	4	50

Iterate through array elements using pointer ptr incrementation

Elem Addr(ptr)	Contents (*ptr)
----------------	-----------------

0x7ffe79199540	10
0x7ffe79199544	20
0x7ffe79199548	30
0x7ffe7919954c	40
0x7ffe79199550	50

Arithmetic on pointers (6)

Two arithmetic operations can be performed on pointers:

2. Subtraction of Pointers:

- **ptr1** and **ptr2** are two pointers of the same type **TypeElement**.
- **ptr1-ptr2** returns an integer:
 - It represents the number of elements separating both slots of the memory.
- The difference between both addresses pointed by **ptr1** and **ptr2** is calculated:
 - It represents the number of bytes, divided by the size of **TypeElement** to obtain the number of elements.

Arithmetic on pointers (7)

Example 1:

```
void main(){
    short int Tab[3]={1 2, 4, 8};
    short int *p;
    p = &Tab[0];
    p++;
}
```

```
void main(){
    short int Tab[3]={1 2, 4, 8};
    short int *p;
    p = &Tab[0];
    p--;
}
```

@	Variable	Value
0x144	??	??
0x145	??	??
0x146	Tab[0]	12
0x147		
0x148	Tab[1]	04
0x149		
0x150	Tab[2]	08
0x151		
0x152	p	0x146
0x153		
0x154	??	??
0x155	??	??
0x156	??	??
0x157	??	??

Arithmetic on pointers (8)

Example 2:

@	Variable	Value
0x144	??	??
0x145	??	??
0x146	Tab[0]	12
0x147		
0x148	Tab[1]	04
0x149		
0x150	Tab[2]	08
0x151		
0x152	p	0x146
0x153		
0x154	??	??
0x155	??	??
0x156	??	??
0x157	??	??

```
void main(){
    short int Tab[3]={12, 4, 8};
    short int *p;
    p = &Tab[0];
    p++;
}
```

@	Variable	Value
0x144	??	??
0x145	??	??
0x146	Tab[0]	12
0x147		
0x148	Tab[1]	04
0x149		
0x150	Tab[2]	08
0x151		
0x152	p	0x148
0x153		
0x154	??	??
0x155	??	??
0x156	??	??
0x157	??	??

Arithmetic on pointers (9)

Example 3:

@	Variable	Value
0x144	??	??
0x145	??	??
0x146	Tab[0]	12
0x147		
0x148	Tab[1]	04
0x149		
0x150	Tab[2]	08
0x151		
0x152	p	0x146
0x153		
0x154	??	??
0x155	??	??
0x156	??	??
0x157	??	??

```
void main(){
    short int Tab[3]={12, 4, 8};
    short int *p;
    p = &Tab[0];
    p--;
}
```

@	Variable	Value
0x144	??	??
0x145	??	??
0x146	Tab[0]	12
0x147		
0x148	Tab[1]	04
0x149		
0x150	Tab[2]	08
0x151		
0x152	p	0x144
0x153		
0x154	??	??
0x155	??	??
0x156	??	??
0x157	??	??

Arithmetic on pointers (10)

Example 4:

```
void main(){
    short int Tab[3]={12, 4, 8};
    short int *p1, *p2;
    p1 = &Tab[2];
    p2 = &Tab[0];
    printf("p1-p2=%d", p1-p2);
}
```

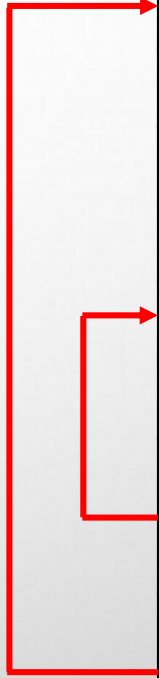
p1 = 0x150

p2 = 0x146

$$p1 - p2 = \frac{0x150 - 0x146}{2} = \frac{4}{2} = 2$$

sizeof(short int)=2

@	Variable	Value
0x144	??	??
0x145	??	??
0x146	Tab[0]	12
0x147		
0x148	Tab[1]	04
0x149		
0x150	Tab[2]	08
0x151		
0x152	p1	0x150
0x153		
0x154	p2	0x146
0x155		
0x156	??	??
0x157	??	??



Arithmetic on pointers (11)

Example 5: Iterate through an array

- It is possible to browse an array **Tab** using a pointer **p** (initialized with the address of the first element of the array **&Tab[0]**) and the expressions **Tab[i]**, ***(p+i)** or ***(Tab+i)**:

```
void main(){
    short int Tab[10];
    short int *p;
    short int i;
    p = &Tab[0];
    for(i=0; i<10; i++)
        printf("element %d: %d", i, *(p+i));
}
```

```
void main(){
    short int Tab[10];
    short int *p;
    short int i;
    p = &Tab[0];
    for(i=0; i<10; i++){
        printf("element %d: %d", i, *p);
        p++;
    }
}
```

Tab always denotes the first element address of the array; whereas, **p** can denote any element.

Arithmetic on pointers (12)

Iterate through an array

- It is possible to use array syntax (`[]` operator) with pointers
 - Expression `p[i]`, equivalent to `*(p+i)`, allows accessing the element located `i` elements after `p`.
- The difference with arrays is that pointers support using negative indices.
 - If (`i < 0`), expression `p[i]` is equivalent to `*(p-i)` to allow accessing the element located `i` elements before `p`.
 - A negative index would not make sense for an array.
 - Array overflow.
- Element of an array may be accessed in different ways:
 - `Tab[i]`.
 - `*(p+i)`.
 - `*(Tab+i)`.
 - `p[i]`.

Arithmetic on pointers (13)

Exercise 1: What values/addresses do these expressions provide:

```
void main(){  
    short int Tab[10]={12, 23, 34, 45, 56, 67, 78, 89, 90, 95};  
    short int *ptr;  
}
```

Expression	Value/Address
ptr=Tab;	
*ptr+2;	
*(ptr+2);	
&ptr+1;	
&Tab[4]-3;	
Tab+3;	
&Tab[7]-ptr;	
ptr+(*ptr-10);	
(ptr+(ptr+8)-Tab[7]);	

Pointers, arrays and strings (1)

Pointers, arrays and strings

- In C/C++, pointers are used to declare arrays and strings, especially when their size is unknown in advance.
 - This is **Dynamic allocation**.

Desig.	Algorithm	C/C++		
Array	const MAX = 100; Tab: Array [MAX] of Integer;	const int MAX=100; int Tab[MAX];	int *T;	int T[];
String	const MAX = 100; Tab: Array [MAX] of Integer;	char str[MAX];	char *str;	char str[];

Pointers, arrays and strings (2)

Pointers, arrays and strings

- Declare an array **Tab**:
 - `TypeElement Tab[MAX];`
- Create a variable **size**:
 - **size** = `MAX * sizeof(TypeElement)` bytes.
- Identifier **Tab** represents a constant expression:
 - **Type**: Pointer to **TypeElement** (i.e., **TypeElement***).
 - **Address**: Address of the 1st element of the array (i.e., `&Tab[0]`).

Tab

Index	0	1	2	3	4	5	6	...	MAX-1
Value	v0	v1	v2	v3	v4	v5	v6	...	
Address	&Tab[0]	&Tab[1]	&Tab[2]	&Tab[3]	&Tab[4]	&Tab[5]	&Tab[6]	...	&Tab[MAX-1]

Pointers, arrays and strings (3)

- Two types of operations may be performed on pointers:
 - **Using & operator:** Manipulate the address contained in the pointer.
 - **Using * operator:** Manipulate the value saved at the pointed address.
- **Modify the value saved at the pointed address:**
 - Realized like for a classic variable
 - With addition of the * operator.
- **Pointed addresses can be modified in different ways:**
 - **Assignment:** already used (\leftarrow).
 - **Addition or subtraction:** Arithmetic on pointers (+, -).

Pointers, arrays and strings (4)

These are character arrays (char[] or char*) where the end of the string is the NULL character ('\0')

Examples:

```
char TabC[] = "abc"; // or char TabC[] = {'a', 'b', 'c', '\0'};  
    // TabC is an array of 4 chars initialized by 4 chars of string "abc"  
    // TabC[0]='a'; TabC[1]='b'; TabC[2]='c'; TabC[3]='\0' ;  
  
char *ptr = "abc";  
    // ptr is initialized with (just) the start address of the string
```

- libc.h
- string.h

Multidimensional arrays (1)

Pointers can also be used to manipulate multidimensional arrays

- A n -dimensional ($n > 1$) array is an array of $(n-1)$ -dimensional arrays

Example:

```
int mat[2][3] = {{1,2,3}, {4,5,6}}; // or {1,2,3,4,5,6}
```

- **mat**: Address of a memory area containing the array elements
 - Arranged row by row.
- Size: $6 \text{ int} = 6 * 4 = 24 \text{ bytes}$:
 - From address `0x100` to address `0x123`).
- **mat[i][j]**: element of row i and column j .

Address	Element	Value
0x100	0, 0	1
0x104	0, 1	2
0x108	0, 2	3
0x112	1, 0	4
0x116	1, 1	5
0x120	1, 2	6


Multidimensional arrays (3)

Pointers can also be used to manipulate multidimensional arrays

Example: Two-dimensional array (2 rows and 3 columns):

```
int mat[2][3] = {1,2,3,4,5,6} ;  
int (*ptr)[3]; // ptr is a pointer to an array of 3 int  
// ptr can therefore be seen as an array of array of 3 int
```

```
ptr = mat;  
// *(ptr+1) (ptr[1], mat[1]) represents the array of the  
// 2nd row of mat, i.e. {4,5,6}  
// (*(ptr+1))[0] (ptr[1][0], mat[1][0]) represents the 1st  
// element of the 2nd row (mat[1][0]), i.e. 4
```



@	Variable	Value
	ptr	0x100
	??	??
	??	??
	??	??
0x100	mat[0][0]	1
0x104	mat[0][1]	2
0x108	mat[0][2]	3
0x112	mat[1][0]	4
0x116	mat[1][1]	5
0x120	mat[1][2]	6
0x124	??	??


Multidimensional arrays (4)

Pointers can also be used to manipulate multidimensional arrays

Example: Two-dimensional array (2 rows and 3 columns):

```
int mat[2][3] = {1,2,3,4,5,6} ;  
int (*ptr)[3]; // ptr is a pointer to an array of 3 int  
// ptr can therefore be seen as an array of array of 3 int
```

```
ptr = mat+1;  
// *ptr (ptr[0], mat[1]) represents the array of the 2nd row  
// of mat, i.e. {4,5,6}  
// (*ptr)[0] (ptr[0][0], mat[1][0]) represents the 1st element  
// of the 2nd row (mat[1][0]), i.e. 4
```



@	Variable	Value
	ptr	0x112
	??	??
	??	??
	??	??
0x100	mat[0][0]	1
0x104	mat[0][1]	2
0x108	mat[0][2]	3
0x112	mat[1][0]	4
0x116	mat[1][1]	5
0x120	mat[1][2]	6
0x124	??	??

Multidimensional arrays (2)

Pointers can also be used to manipulate multidimensional arrays

To access the element at $[i][j]$ in an array of **nLine** rows and **nCol** columns, the compiler performs the following calculation:

$$@[i][j] = @base + \text{sizeof}(\text{TypeElement}) * (i * \text{nCol} + j)$$

In the above example, the address of:

$$\text{mat}[1][1] = 0x100 + 4 * (1 * 3 + 1) = 0x116$$

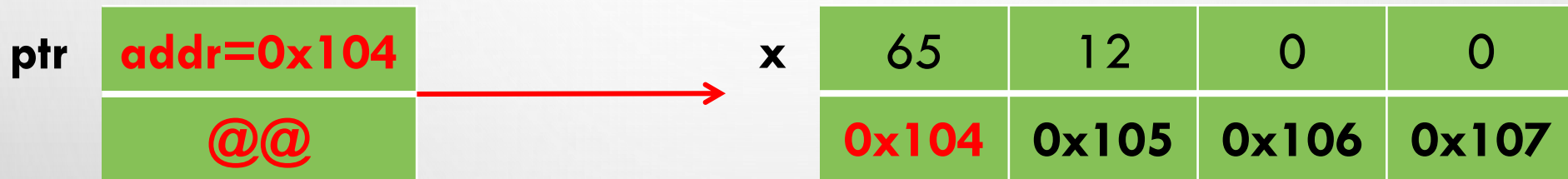
In case of a k-dimensional array: $\text{Tab}[n_1][n_2] \dots [n_k]$ is:

$$@[i_1][i_2] \dots [i_k] = @base + \text{sizeof}(\text{TypeElement}) * (i_1 * n_2 * n_3 * \dots * n_k + i_2 * n_3 * n_4 * \dots * n_k + \dots + i_{k-1} * n_k + i_k)$$

Dereferencing a pointer

In C/C++, Dereferencing a pointer (using type cast)

- If a pointer **ptr** contains an address **addr**:
 - ***(Type*)ptr** is considered as a variable (of type **Type**).
 - Located at address **addr**.



- **0x104** is the address of a Char variable:
 - `printf("%c", *(char*)ptr)`
 - **'A'** (65).
- **0x104** is the address of an Integer variable:
 - `printf("%d", *(int*)ptr)`
 - **Value = ?? (How much?)**

References

- Akinocho Ghislain, “Algorithmique et Structures de données”, France, 2010.
- Walid Khaled Hidouci, “Basics: Pointers and memory allocation in C”, National Higher School of Computer Sciences, Algiers.
- Sana Aroussi, “Algorithmique et Structures de données dynamiques 2”, University Saad Dahleb of Blida 1, 2023/2024.
- Walid Khaled Hidouci, “Basics: The C programming language: A very brief introduction”, National Higher School of Computer Sciences, Algiers.
- Mohamed Messabihi, “Initiation à l’algorithmique: Les pointeurs”, University of de Tlemcen.
- Damien Berthet, Vincent Labatut, “Algorithmique & programmation en langage C (Vol. 1): Supports de cours”, Istanbul, Turkey, 2014.



Thank You!

abdelfetah.hentout@enscs.edu.dz